VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH™
SOFTWARE SYSTEM
for ATARI®

PLAYER-MISSILE GRAPHICS

Software and Documentation
©Copyright 1982
Valpar International

®Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH™
SOFTWARE SYSTEM

PLAYER-MISSILE GRAPHICS

Stephen Maguire

Software and Documentation
©Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

valFORTH T.M.

PLAYER-MISSILE GRAPHICS

Version 1.0
April 1982

The following is a description of commands used in creating seemingly difficult video displays using players and missiles. Used alone or in combination with the other available systems by Valpar International, it is possible to obtain graphic displays which compare with those of the best arcade games. The use of players and missiles (also called "player/missiles") allows the beginner to create high quality moving video displays.

Software and Documentation
©Copyright 1982
Valpar International

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

PLAYER/MISSILE GRAPHICS PACKAGE

XXI. PLAYER/MISSILE GRAPHICS

a) STROLLING THROUGH PLAYER/MISSILE GRAPHICS 1
b) PLAYER/MISSILE GLOSSARY
1) ENABLING PLAYER/MISSILE GRAPHICS 6
2) CREATING PLAYERS AND MISSILES 10
3) MOVING/PLACING PLAYERS AND MISSILES 12
4) SETTING PLAYER/MISSILE BOUNDARIES 14
5) COLLISIONS BETWEEN PLAYERS AND MISSILES 16

XXII. CHARACTER EDITOR
User's manual for the character set editor.

XXIII. SOUND EDITOR
Description of the audio-palette sound editor.

XXIV. PLAYER/MISSILE SUPPLIED SOURCE

As knowledge of the internal workings of player/missile graphics is not necessary to use this valFORTH package effectively, the internal workings are not explained in this manual. However, for the serious programmer trying to optimize his/her program in every way, an understanding of these internal workings could at times improve code efficiency and/or speed of execution. For a complete explanation of player/missile graphics at the nut-and-bolt level, see the series of articles by Dave and Sandy Small in Creative Computing.

STROLLING THROUGH PLAYER/MISSILE GRAPHICS

One of the biggest differences between the Atari graphic capabilities and those of most other computers is the Atari's ability to use players and missiles. This discussion will not explain the internal workings of player/missile graphics on the Atari; rather, it will explain how to use the basic commands in this valFORTH package. Before we proceed, please load the player/missile graphic routines from the Player/Missile disk. The directory on screen 170 will show what screen to load. Also, if you have the valFORTH Editor/Utilities package, load in the high speed STICK command found in the Miscellaneous Utilities; otherwise, load in the slower version on your Player/Missile disk. (Check the directory for its location).

To start with, let's get a simple player up on the screen to experiment with. First we must initialize the player/missile graphic system and design the player's image. This is simple:

```
1  PMINIT                    ( Initialize for single
                               resolution players )

2  BASE  !                   ( Change to binary for ease )

LABEL CROSS                  ( Give the player image a name )
00011000  C,
00011000  C,
00011000  C,
11111111  C,                 ( A large plus sign )
11111111  C,
00011000  C,
00011000  C,
00011000  C,

DECIMAL                      ( Now back into base 10 )

PMCLR                        ( Clear player/missile memory )

ON PLAYERS                   ( Turn on the players )

CROSS  8  180  50  0  BLDPLY      ( Build a player )
```

You should now see the cross in the upper right-hand corner of the video screen. Now let's take a look at this and see how it works.

First, players are initialized using the PMINIT command. Players can be in either a single or double resolution mode (double res players are twice as tall). "1 PMINIT" is used for single res players. If we had wanted double res players, we would have used "2 PMINIT".

Next, the player image is created. Since it is much easier to make player images as 1's and 0's, we use binary (base two) number entry. Before we design the image, it must be given a name. The LABEL command does this nicely for us.

This image is named CROSS. All that need be done now is to draw the picture. Notice how easy it is to see the image when using base two. Of course, we could have stayed in base 10 and still designed the image, but this is usually more difficult. The word C, after each number simply tells FORTH to store that number in the dictionary. Once the picture is designed, we return to decimal for ease.

Both the PMCLR and ON PLAYERS commands are fairly self-descriptive: PMCLR erases all players and missiles so that no random trash appears when the PLAYERS are turned ON. Next, the BLDPLY (build player) command takes the image named CROSS which is 8 bytes tall and assigns it to player 0 at horizontal location 180 and vertical location 50 on the display. Of course, we could have built player 1, 2, or 3 instead.

The cross should be black. Suppose we wanted a blue or green cross instead. This can be done using the PMCOL (player/missile color) command. Try this:

```
0  9  8  PMCOL              ( player  hue  lum  PMCOL )
```

The cross should now appear blue. This command assigns a BLUE (9) hue with a luminance of 8 to player 0. If the color commands are loaded from the valFORTH disk,

```
0  BLUE  8  PMCOL
```

could have been used with the same results. Try changing the color of the player to GREEN (12) or PINK (4). Note that the default colors for players 2 and 3 make them invisible: Their colors should be set immediately upon being built.

Now that we have a player on the screen, let's move it around. We use the PLYMV (player move) command for this. PLYMV needs to know which player to move (there could be as many as five), how far to move it in the horizontal direction, and how far to move it in the vertical direction. Try this:

```
1  1  0  PLYMV            ( horz  vert  player  PLYMV )
```

This moves player 0 down 1 line and right one horizontal position, thus giving the effect of a diagonal move towards the lower right-hand corner. Try these as well:

```
 1   0  0  PLYMV           ( move right one position )
-5   0  0  PLYMV           ( move left five positions )
 0  20  0  PLYMV           ( move down 20 lines )
 0 -15  0  PLYMV           ( move up 15 lines )
-5   2  0  PLYMV           ( move left five, and down two )
```

That's all there is to moving a player. Positive horizontal offsets move the player right, and negative values move the player left. Likewise, positive vertical offsets move the player down while negative ones move the player up. The following program can be typed in and you will have a joystick controlled player:

```
: JOY
  BEGIN
    0 STICK                ( STICK leaves two offsets )
    0 PLYMV                ( for PLYMV to use. )
    ?TERMINAL
  UNTIL ;
```

JOY <ret>

Move the player with stick 0, the left-most stick port. Press any console button to exit the program.

Currently, if the player is moved off any edge, it "wraps" to the opposite side. In other words, we have an "unbound" player. This is rarely desirable. Normally, we want to restrict player movement to certain boundaries. The PLYMV command has a built in boundary check routine specifically for this reason. Right now, new boundaries are set so wrapping occurs. Let's set some boundaries:

```
60  150  50  200  0  PLYBND
```

This sets the boundaries of player zero to 75 on the left, 150 on the right, 50 on the top, and 200 on the bottom. Type JOY again to verify that you can no longer move freely about the display. Try different boundary settings and experiment to get the feel of the command. Boundary checking can be disabled for any or all of the edges. Setting the left or upper boundary to 0 will disable the check on that edge, likewise, 255 in either the right or lower boundary will do the same.

Let's build another player in the lower right-hand corner of the screen. This time, instead of designing the player ourself, let's borrow the image from the standard Atari character set stored in ROM. The image of the digit zero starts at address 57472. The other numbers follow zero. Try this:

```
57472  16  160  150  1  BLDPLY
```

You should now see the numbers 0 and 1 on your screen. This command builds player 1 with the image at address 57472 that is 16 bytes tall and puts it at horizontal position 160 and vertical position 150. Give this player a color if you want.

Until now, we have been using normal size players. It is possible to make the two players on the display different widths using the PLYWID command. PLYWID expects a width specification of 0 or 2 (normal), 1 (double), or 3 (quadruple). Its command form is:

```
width   player   PLYWID
```

Thus,

```
3  1  PLYWID
```

should make player one four times its original size. The same can be done with player zero:

```
3  0  PLYWID
```

Type JOY again and notice that the width has no effect on movement whatsoever. Also notice that player one is unaffected by movement of player zero.

Now that we have two players on the screen, let's interface both of them to the joystick. Type in the following program:

```
: JOY2
  BEGIN
    0  STICK                ( Record stick movement )
    2DUP                    ( Make a copy )
    0  PLYMV                ( Move player 0 )
    SWAP                    ( Rotate stick 90 degrees )
    1  PLYMV                ( Move player 1 )
    ?TERMINAL
  UNTIL ;

JOY2  <ret>
```

Notice that when you push the stick up, player zero goes up, but player one moves left. The SWAP instruction exchanges the vertical and horizontal offsets from STICK before moving player one. If we were to take the SWAP out, the players would move identically.

In many applications, it is necessary to know when a player has hit another player or some background image. Fortunately, the Atari computer automatically makes this information available. An entire collection of valFORTH words allows checking of all collisions possible. The most general word is ?COL which simply returns a true flag if anything has hit anything else. Here is an example:

```
: BUMP
  BEGIN
    HITCLR
    0  STICK
    0  PLYMV
    ?COL
    IF
      CR  ." oops!"
    ENDIF
    ?TERMINAL
  UNTIL  ;

BUMP  <ret>
```

Move the player around and watch the results. Every time you hit any letters or player one, the word "oops!" should be printed out. This program is quite simple. First, the HITCLR command is issued which erases any old collision information. If this command were omitted, the first time a collision occurred, "oops!" would be continuously printed out. Next the joystick is read and the player moved. If the player touches anything when moved, the collision registers are set. ?COL reads these registers and leaves a true flag if the player has hit something, and the IF statement will then print out "oops!".

Using other commands found in the glossary, we can tell specifically what the player has hit. For example, the ?PXPF command checks to see if a specific player has hit a playfield, and if so, it returns information indicating which playfield.

Although this discussion was limited to using players, the routines for missiles function similarly and can be found in the following glossary. Two player/missile example programs can be found on your Player/Missile disk. These demonstrate how short player/missile routines can be.

PLAYER/MISSILE GLOSSARY

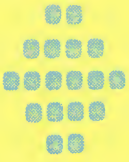
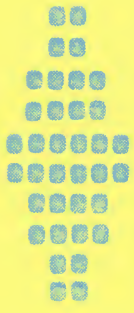
Enabling Player-Missile Graphics

To make use of players and missiles, the video processor must be activated. Players can be several sizes, they can have different overlap priority schemes, and they can have different colors. The following collection of "words" makes this setup task quite simple. Note: Players and missiles are numbered 0 through 3. The fifth player is numbered as four.

(PMINIT) (addr res ---)

The (PMINIT) command (or PMINIT below) must be used to initialize the player missile routines before any other player missile command may be used. (PMINIT) expects both the address of player/missile memory and a 1 or a 2 indicating whether single or double resolution is desired.

NOTE: The difference between single and double resolution is shown graphically below:

Player as defined in memory:	single res on screen:	double res on screen:
00011000		
00111100		
01111110		
00111100		
00011000		

PMINIT (res ---)

The PMINIT command functions identically to the (PMINIT) command above, except that no address need be given. PMINIT calculates an address based on the current graphic mode. It uses the first unused 2K block of memory below the highest free memory (i.e., below the display list). This should only be used while first learning the system, after that, (PMINIT) should be used to optimize memory utilization. Note that the variable PMBAS contains the calculated address upon return.

PMBAS (--- addr)

A variable containing the address of player/missile memory. This value must lie on a 2K boundary if single resolution players are used and on a 1K boundary if double resolution players are used. This is set using the (PMINIT) command and is automatically set by the PMINIT command described above. This value should never be set directly, but can be read at any time.

PLAYERS (ON/OFF ---)

If the flag found on the top of the stack equates to TRUE or ON, then the player/missiles are activated. This does not clear out player missile memory; therefore, the PMCLR command described below is usually used prior to enabling the players and missiles to ensure that no random trash appears on the screen.

If the flag found on the top of the stack equates to FALSE or OFF, then the player/missile graphic mode is de-activated. Turning players off does not clear player-missile memory; therefore, a subsequent ON PLAYERS command would redisplay any previously defined players and missiles. If players are already disabled, the command is ignored.

5THPLY (flag ---)

In many applications it is desirable to combine the four missiles and simulate a fifth player, thus giving five players (numbered 0-4), and no missiles. If the flag on the stack is non-zero, then the fifth player mode will be initiated; otherwise, the missile mode will be re-activated.

Normally, missiles take on the color of their corresponding players; however, when a fifth player is asked for, all missiles take on the common color of playfield #3. In addition, it also allows the fifth player to be treated exactly as any other player would be treated. Bear in mind that although it is called a "fifth" player, its reference number is four (4). The fifth player is "built" with missile zero on the right, and missile three on the left:

|m3|m2|m1|m0| - fifth player

(Note: For convenience, the words ON and OFF have been defined to allow niceties such as:

ON 5THPLY
OFF 5THPLY

These two words are recognized by all words that require an ON/OFF type indication.)

PLYCLR (pl# ---)

Few applications use all available players. To keep these unused players from displaying trash, they can be cleared of all data by using the PLYCLR command. The PLYCLR command expects the player number on the top of the stack and fills the specified player with zeroes. This command can be used to "turn off" players which are no longer needed.

MSLCLR (ml# ---)

The MSLCLR command is very much like the PLYCLR command, described above, except that it clears the specified missile. In addition, this can be used when the fifth player is activated to erase parts of the fifth player for special effects.

PMCLR (---)

This command clears all players and all missiles. This is generally used just prior to activating the player-missile graphic mode to ensure that no random trash is placed on the video screen. PMCLR expects no values on the stack, nor does it leave any.

MCPLY (F ---)

The MCPLY (Multi-Color Player) command expects one value on the top of the stack. If this value is 0 or OFF, then the multi-color player mode is disabled. If this value is 1 or ON, this command instructs the video processor to logically "or" the bits of the colors of player zero with player one, and also of player two with player three. In other words, when players 0 and 1 overlap (or players 2 and 3), a third color (determined by the colors of the overlapping players) will be assigned to the overlapped region rather than assigning one of the players a higher priority. Since players must be one color, this allows for multi-colored players. For example:

```
  Player 0          Player 1          MCPlayer
 Pink color        Blue color       Pink/blue
   ( 4 )             ( 8 )          ( 4 OR 8
                                     = green )

                     BBBB              BBBB
                   BBBBBBBB          BBBBBBBB
 PPPPPPPP                            PPPPPPPP
 PPPPPPPP          BB  BB            PGGPPGGP
 PPPPPPPP                            PPPPPPPP
  PP  PP                              PP  PP
   PPPP                                PPPP
```

NOTE: The lums of the two players are also OR'd.

PRIOR (n ---)

The PRIOR command expects one value on the top of the stack. This value must be 8, 4, 2, or 1, otherwise unpredictable video displays may occur. PRIOR instructs the video processor as to what has higher priority for a video location on the screen. For example, it will determine whether a plane (a player) will pass in front of a building (a playfield), or whether the plane will pass behind the building. Objects with higher priorities will appear to pass in front of those with lower priorities. The following table shows the available priority settings:

n=0	n=4	n=2	n=1
PF0	PF0	PL0	PL0
PF1	PF1	PL1	PL1
PL0	PF2	PF0	PL2
PL1	PF3*	PF1	PL3
PL2	PL0	PF2	PF0
PL3	PL1	PF3*	PF1
PF2	PL2	PL2	PF2
PF3*	PL3	PL3	PF3*
BAK	BAK	BAK	BAK

* PF3 and PL4 share the same priority

Objects higher on the list will appear to pass in front of objects lower on the list.

CREATING PLAYERS AND MISSILES

Once the player/missile graphics system has been activated and the priorities set, all that need be done is to create the players themselves. Normally, this would be quite difficult to do; however, using the commands and designing techniques described below, this task is made very simple. There are really only three things to do in the creation of a player: setting the width size, setting the color, and creating the picture.

PLYWID (width pl# ---)

The PLYWID command sets the specified player to the desired width. Players are numbered 0, 1, 2, 3, or in the case of the fifth player, 4. Legal widths are:

image: 10111101

0 = normal width:
1 = double width:
2 = normal width:
3 = quad. width:

Any other value may cause strange results.

MSLWID (size ml# ---)

The MSLWID command is identical to the PLYWID command described above except that it is used to set the size of the missiles. The same size values apply also. The MSLWID command should only be used when in the missile mode (i.e., with the fifth player deactivated).

PMCOL (pl# hue lum ---)

To set the color (hue and lum) of a player, the PMCOL (Player-Missile-Color) command is used. It sets the specified player to the hue and lumina desired. Note that there is no corresponding command to set the colors of missiles as missiles take on the colors of their respective players. To set the color of the 5th player, "pl#" should be 4. If the color words on the valFORTH 1.1 disk are loaded, they can be used to set player colors:

```
0 BLUE 8 PMCOL
```

This sets player #0 to a medium blue color.

BLDPLY (addr len horz vert pl# ---)

The BLDPLY command is probably the most useful of all the commands in this graphic package. It takes an easily predefined picture that resides in memory at address "addr" whose length is "len" and converts it to the specified player "pl#". It then positions the player at the coordinates (horz,vert). The player is then ready to be moved about the screen using the PLYMV command described below.

As an example, a player in the form of an arrow pointing upward will be created, assuming that priorities and such have already been taken care of. Practice has proven that the following method is easiest for creating players:

```
2 BASE  !           ( put into binary mode   )

LABEL PICTURE       ( the image is named PICTURE )
  00011000 C,
  00111100 C,
  01111110 C,
  11011011 C,
  00011000 C,
  00011000 C,
  00011000 C,
  00011000 C,
DECIMAL

1 PMINIT            ( initialize for single resolution )
PICTURE  8  80  40  0  BLDPLY
```

Takes the image at location PICTURE which is 8 bytes long, and builds player #0 at location (80,40).

BLDMSL (addr len horz vert m1# ---)

The BLDPLY command described above does just about everything necessary to create a high-resolution player. The BLDMSL command functions identically to the BLDPLY command except that it is used for setting up missiles (which are in effect just skinny players). The method for creating players can be used for creating missiles as well. Note that if the fifth player mode is activated, the BLDPLY command must be used to create the player.

Building missiles takes a bit more care than building players. Players occupy separate memory, while the four missiles share the same memory. Each missile is two bits wide; all four together are exactly a byte wide. Missile memory is shared with the two lowest bits devoted to missile zero, and the two highest bits devoted to missile three:

| m3 | m2 | m2 | m2 | m1 | m1 | m0 | m0 |

All players with the same shape can use the same image without any problem since they all are a full byte wide. Missiles, however, cannot use the same shape since their images must be ORed into missile memory. This means that the missile images must be in the proper bit columns. For example, the same image for separate missiles could be:

```
11000000   00110000   00001100   00000011
11000000   00110000   00001100   00000011
11000000   00110000   00001100   00000011
  msl#3      msl#2      msl#1      msl#0
```

PUTTING PLAYERS AND MISSILES IN THEIR PLACE

Generally, once a player or missile has been created and put to the video screen, it is moved around. This can be accomplished very easily with the next set of words. Interfacing a movable player with the joystick can improve just about any program which requires input. As a result, it usually gives the program a more professional appearance.

PLYLOC (pl# --- horz vert)

The PLYLOC command (PLaYer LOCation) returns the vertical and horizontal positions of the specified player. This is normally used when a joystick/button setup is being utilized -- i.e., when a joystick is moving a player and the button is used to pinpoint where the player is. A program which draws lines between two dots could use this. The joystick is used to move the player to the desired spot on the screen. Pressing the button tells the program that a selected spot has been made. Once a second spot has been selected, the program then draws a line between them.

MSLLOC (ml# --- horz vert)

The MSLLOC command performs the same function as the PLYLOC command described above except that it is used to find locations of missiles instead of players. Note that using MSLLOC on a fifth player gives meaningless results.

PLYMV (horz vert pl# ---)

The PLaYer MoVe command moves the specified player the direction specified by "vert" and "horz". If "vert" or "horz" is negative, the player is moved up or left respectively, otherwise it is moved down or right unless they happen to be zero in which case nothing happens. The following examples clarify this:

```
 0 -5  0 PLYMV    ( Move player 0 up 5 lines )
-1 -1  3 PLYMV    ( Move player 3 left and up one line )
 3 -1  2 PLYMV    ( Move player 2 up one dot and right 3 )
```

MSLMV (horz vert ml# ---)

The MSLMV is identical in function as the PLYMV command described above except that it is used to move missiles about the video screen.

PLYPUT (horz vert pl# ---)

The PLYPUT command positions player "pl#" to the location (horz,vert) on the video screen.

PLYCHG (addr len pl# ---)

Oftentimes it is necessary to change the image of a player after it has been built. The PLYCHG command allows this to be easily done. The PLYCHG command takes the image with length "len" at address "addr" and assigns it to player 'pl#". Note that if the new image is shorter than the previous one, part of the previous image will remain. This can be overcome by executing a PLYCLR command prior to PLYCHG.

PLYSEL (addr # pl# --)

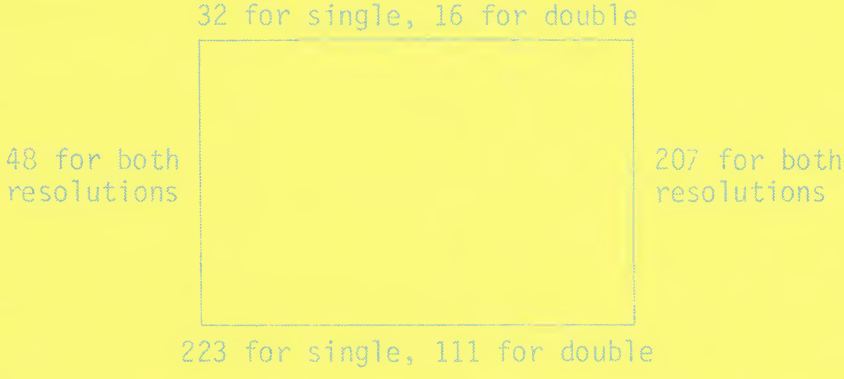
The PLYSEL command is used to select image "#" out of a table of images of the same length and assigns that image to the specified player. PLYSEL is typically used to animate players. An example usage of this can be found in Player/Missile Example #2 found in the directory of the disk.

PLAYER/MISSILE BOUNDARIES

It is often desirable to put limitations on the movements of players and missiles. Boundaries can be set up for each player and missile independently and upon each move command, they will remain within those boundaries. Additionally, a boundary status byte for each player is available for scrutiny at any time. This section explains how this is used.

PLYBND (left right top bottom pl# --)

In most applications, the movements of players are kept within certain boundaries. The PLYBND command frees the user from having to worry about boundary checking. This command expects the player number and all four boundaries. Whenever a PLYMV is then used, the player is always kept within the set boundaries. Also, upon each move a boundary status byte is left in the c-array PLYSTT (see ?PLYSTT below). The edge boundaries of the screen are:



Note that in special cases the boundary checker will fail. If the left boundary is 0 and the player is at the boundary, any move left will not be checked as expected. For example, if it were moved left by one position (-1), the new horizontal position would be -1 or FFFF in hex. Since only 8 bit unsigned comparisons are made, the horizontal position appears to be 255 (FF hex). Post calculating boundary checking turns out to be more useful because it allows any or all edges to be unbounded. If an unbounded player is desired, use this:

```
0  255  0  255  pl#  PLYBND
```

For an example of PLYBND, see the example program found in the directory on screen 170 of your disk.

MSLBND (left right top bottom ml# --)

The MSLBND command is the same as the PLYBND command above, except that it is used for missiles. Upon each move a boundary status byte is left in the array MSLSTT. See ?MSLSTT below.

?BND (--- n)

This command leaves the boundary check status of the last PLYMV or MSLMV performed. The value has the following form:

Only the lower four bits are of use. Each bit represents a different edge. If the bit is set, then the player or missile has attempted to move beyond that boundary. Note that only two of the four bits can be set at any time.

Note: DECIMAL
```
...
?BND  3  AND
IF  hit-vertical-boundary ENDIF
?BND  12  AND
IF  hit-horizontal-boundary ENDIF
...
```

?PLYSTT (pl# --- val)

Given a player number, returns the boundary check byte of that player. This byte is the status byte for the most recent PLYMV of that player. See ?BND above for the description of the status byte.

?MSLSTT (ml# --- val)

Given a missile number, returns the boundary check byte of that missile. This byte is the status byte for the most recent MSLMV of that missile. See ?BND above for the description of the status byte.

CHECKING FOR INTERACTION BETWEEN PLAYERS

All the commands given so far allow the creation of any player or missile desired. But once that player is on the screen and moving around, it is often necessary to know when two or more objects (players, missiles, and playfields) touch or "crash" into each other. This remaining collection of commands allows checking of all possible "hit" combinations.

?COL (--- f)

The ?COL command is a very general collision detector. It does nothing more than indicate whether two or more objects have "crashed" -- it does not give any indication of what has collided. It leaves a 1 on the stack if a collision has taken place; otherwise it leaves a zero.

?MXPF (ml# --- n)

The ?MXPF command is a much more specific collision detection command. It stands for "?collision of Missile #X with any PlayField". It is used to check if a specific missile has hit any playfield. It returns a zero if no collision has taken place, and leaves an 8, 4, 2, 1, or combinations of these (e.g., 12 = 8+4) if a collision has occurred. Each of these four basic values represents a specific playfield:

3 ?MXPF (Has missile #3 hit any playfields?)

TOS	binary	meaning of val
0	0000	no collisions
1	0001	with pf#0
2	0010	with pf#1
3	0011	with pf#0,1
4	0100	with pf#2
5	0101	with pf#2,0
6	0110	with pf#2,1
7	0111	with pf#2,1,0
8	1000	with pf#3
9	1001	with pf#3,0
10	1010	with pf#3,1
11	1011	with pf#3,1,0
12	1100	with pf#3,2
13	1101	with pf#3,2,0
14	1110	with pf#3,2,1
15	1111	with pf#3,2,1,0

To test for a collision with one specific playfield, use one of the following:

```
1 AND      ( Leaves 1 if collision with pf#0, else 0 )
2 AND      (   "    1       "         "  pf#1,  "   0 )
4 AND      (   "    1       "         "  pf#2,  "   0 )
8 AND      (   "    1       "         "  pf#3,  "   0 )
```

?PXPF (pl# --- n)

The ?PXPF command (?collision of Player #X with any PlayField) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions with players and playfields instead of missiles and playfields.

?MXPL (ml# --- n)

The ?MXPL command (?collision of Missile #X with any Player) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between missiles and players. Note that it is impossible for a missile to collide with a fifth player since it would be, in effect, colliding with itself.

?PXPL (pl# --- n)

The ?PXPL command (?collision of Player #X with any other players) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between players. Note that it is impossible for a player to collide with itself.

HITCLR (---)

The HITCLR command clears all collision registers. In other words, it sets the collision monitor to a state which indicates that no collisions have occurred.

HANDY REFERENCE CARD

valFORTH SOFTWARE SYSTEM

PLAYER-MISSILE GRAPHICS, CHARACTER EDITOR, & SOUND EDITOR

Player/Missile Command Summary

Note: Players and missiles are numbered 0 thru 3. The fifth player is numbered 4.

(PMINIT)	(addr res ---)	Initializes the player missile routines with PM memory specified by "addr" with "res" resolution.
PMINIT	(res ---)	Initializes the player missile routines with "res" resolution and with PM memory located at the first available memory below the display list.
PMBAS	(--- addr)	A variable pointing to player/missile memory which is set by (PMINIT) or PMINIT. It can be read from but not written to.
PLAYERS	(ON/OFF ---)	This command enables or disables the player/missile graphic display
5THPLY	(ON/OFF ---)	This command turns (the fifth player mode) ON or OFF. If OFF, missiles take the colors of their corresponding players. If ON, all missiles take on the common color of playfield 3. The fifth player is numbered as four (4).
PLYCLR	(pl# ---)	Erases the specified player (0-3,4).
MSLCLR	(ml# ---)	Erases the specified missile (0-3).
PMCLR	(---)	Erases all players and all missiles.
MCPLY	(ON/OFF ---)	This command turns (the multiple color player mode) ON or OFF. See documentation for explanation.
PRIOR	(n ---)	Sets the priority of players and playfields. See documentation for legal settings.
PLYWID	(width pl# ---)	Sets the width of the specified player. Legal widths are normal (0 or 2), double (1), or quadruple (3).
MSLWID	(width ml# ---)	Sets the width of the specified missile. Legal widths are normal (0 or 2), double (1), or quadruple (3).
PMCOL	(pl# hue lum ---)	Sets the specified player to the color defined by "hue" and "lum".
BLDPLY	(addr len horz vert pl# ---)	Creates a player whose image is at "addr" with a length "len". The player is originally placed at the specified horizontal and vertical coordinates.
BLDMSL	(addr len horz vert ml# ---)	Creates a missile whose image is at "addr" with a length "len". The player is originally placed at the specified horizontal and vertical coordinates.
PLYLOC	(pl# --- horz vert)	Returns the horizontal and vertical coordinates of the specified player.
MSLLOC	(ml# --- horz vert)	Returns the horizontal and vertical coordinates of the specified missile.
PLYMV	(horz vert pl# ---)	Moves the specified player according to the horizontal and vertical offsets specified. A positive horizontal offset moves the player right, a negative one moves it left. Likewise, a positive vertical offset moves the player down and a negative one moves it up
MSLMV	(horz vert ml# ---)	Moves the specified missile according to the horizontal and vertical offsets specified. See PLYMV above
PLYPUT	(x y pl# ---)	Positions the specified player and location (x,y) on the video display.
PLYCHG	(addr len pl# ---)	This changes the image of the specified player to the image of length "len" at "addr".
PLYSEL	(addr # pl# ---)	This changes the image of the specified player to image number "#" in a table of images starting at address "addr".
PLYBND	(l r t b pl# ---)	Specifies the left, right, top, and bottom boundaries of the specified player.
MSLBND	(l r t b ml# ---)	Specifies the left, right, top, and bottom boundaries of the specified missile.
?BND	(--- n)	Returns the boundary status of the last player or missile moved. See documentation for description of this value.
?PLYBTT	(pl# --- n)	Returns the boundary status of the last move of the specified player. See documentation for a description of this value.
?MSLBTT	(ml# --- n)	Returns the boundary status of the last move of the specified missile. See documentation for a description of this value.
?COL	(--- f)	Returns true (1) if any collisions have occurred since the last HITCLR command was issued.
?MXPF	(ml# --- n)	Returns 0 if the specified missile has not hit any playfields since the last HITCLR command. If any collisions have occurred, a status value is returned. See documentation.
?PXPF	(pl# --- n)	Returns 0 if the specified player has not hit any playfield since the last HITCLR command. If any collisions have occurred, a status value is returned. See documentation.
?MXPL	(ml# --- n)	Returns 0 if the specified missile has not hit any player since the last HITCLR command. If any collisions have occurred, a status value is returned. See documentation.
?PXPL	(pl# --- n)	Returns 0 if the specified player has not hit any other players since the last HITCLR command. If any collisions have occurred, a status value is returned.
HITCLR	(---)	Clears the collision registers to a no-collision state.

Audio Editor Command Summary

AMUSE	(--)	Calls up the audio-palette program.

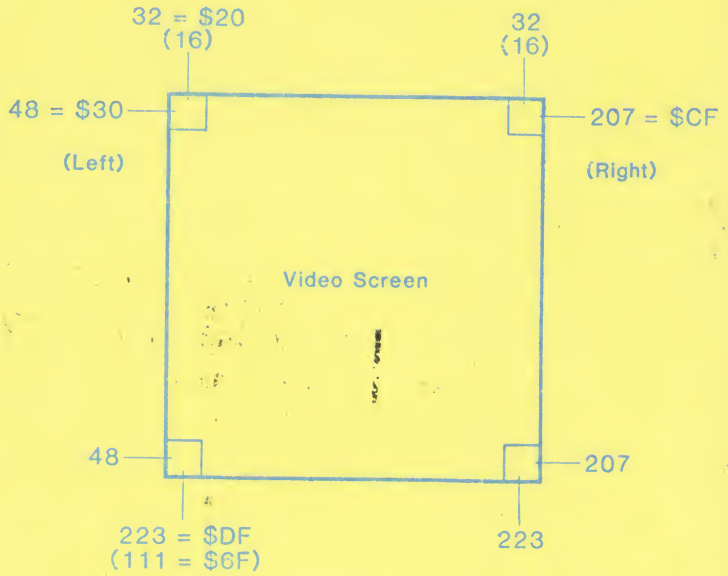
Character Editor Command Summary

CHAR-EDIT	(--)	Calls up the character editor.

PLAYER-MISSILE BOUNDARY MAP

valFORTH

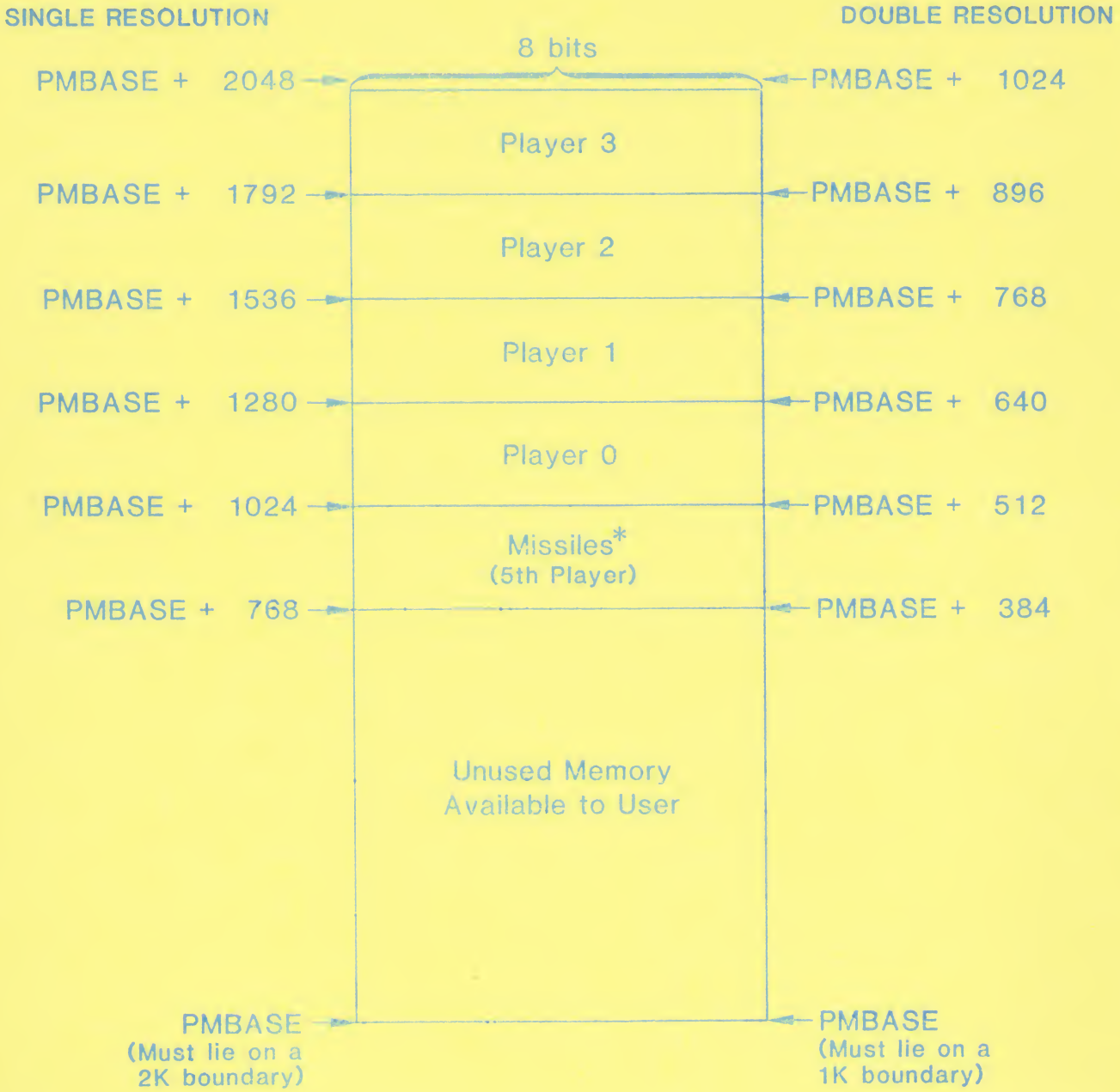
(Double resolution values are in parentheses)



Software and Documentation
©Copyright 1982
Valpar International

PLAYER-MISSILE Memory Map

valFORTH™



*Note: All missiles occupy the same memory location. This is possible because unlike players which are 8 bits wide and fill an entire byte, missiles are only two bits wide. Four missiles can therefore be represented in the same amount of memory as a single player.

Byte form: ⁞ m3 ⁞ m2 ⁞ m1 ⁞ m0 ⁞

©VALPAR INTERNATIONAL

HANDY REFERENCE CARD

valFORTH™

SOFTWARE SYSTEM

DISPLAY FORMATTER

DINIT	(dmem dlist ---)	This command initializes the display formatter using 'dlist' as the target address for the display list, and 'dmem' as the target address for display memory.
MINIT	(---)	This command initializes the display formatter setting the display memory address to top-of-memory minus $1F00. The display list is targeted for $400 bytes below the display memory.
DBM	(opcode ---)	The DBM command adds 'opcode' to the end of the current display list.
LBMS	(#times opcode ---)	The LBMS command performs a multiple DBM command as described above.
DBPTR	(block# ---)	This command makes the specified block the next block to be created with the DBM command. It essentially makes block#-1 the end of the current list.
DBIN	(opcode block# ---)	The DBIN command inserts the specified opcode into the specified block. That block and all following blocks are pushed back one block.
DBINS	(#times opcode block# ---)	This command performs a multiple DBIN command as described above.
DBDEL	(block# ---)	The DBDEL command deletes the specified block from the current display list.
DBDELS	(#times block# ---)	This command performs a multiple DBDEL command as described above.
DBDELL	(---)	The DBDELL command deletes the last block of the current display list.
DBMOD	(modifier block# ---)	This command modifies the specified block. Legal modifiers are VRMOD, HRZMOD, and INTMOD.
DBMODL	(modifier ---)	This command modifies the last block of the current display list.
DBREM	(block# ---)	The DBREM command removes all modifiers from the specified block. Note that if a HRZMOD is stripped, display memory allocation will change.
DBREMS	(#times block# ---)	This command is a multiple DBREM command.
DBMDEL	(block# ---)	This command strips modifiers from the last block of the current display list.
DBVAL	(block# --- value)	The DBVAL command returns all information about the display block specified, i.e., the antic mode and any modifiers. This information is returned as one value.
DBANTMOD	(block# --- antic-mode)	This command returns the antic-mode (or opcode) of the specified block.
DBMODS	(block# --- modifiers)	The DBMODS command returns the modifiers for the specified block. This information is returned as one value. See documentation for notes on interpretation of this value.
DSWID	(width ---)	The DSWID command sets the display formatter up for narrow (1), normal (2), or wide (3) screen display.
DBADR	(block# --- address)	Given a display block number, it returns the address of the first byte of that display block.
DMOVE	(---)	The DMOVE command clears the display memory pointed to by the display list currently being created.
USRDSP	(---)	Once a display list has been created, USRDSP activates the new list.
MIXED	(---)	The MIXED command performs a USRDSP then instructs the Atari operating system to re-assert all options to the video display memory expected by the newly created display list.
DMPLST	(---)	The DMPLST command gives a complete, informative listing of the display list currently being created.
DLPOS	(--- address)	A variable with contains a pointer to the end of the current display list. It is an offset from DISPLST.
DBNUM	(--- address)	A variable containing the number of the next display block to be created.
DBPTR	(--- address)	A variable which contains the target address of the display memory pointed to by the current display list.
LMSADR	(--- address)	A variable which contains the target address for the current display list.
DBLAY	(block# --- address)	An array of addresses from the DBADR
DBTAB	(pointer --- address)	A variable containing the display list currently being created. ADVMD always points to the end of the list at this point.

HANDY REFERENCE CARD

valFORTH™

SOFTWARE SYSTEM

DISPLAY FORMATTER

The Character Modes

There are 6 character modes (opcodes 2 thru 7). All character modes work in the same way, i.e., the values in display memory are indices to a large "n" by 8 byte array. In some of these modes, the highest one or two bits are used to specify a color with only the remaining lower bits used for indexing. The following table gives information about each of the modes.

Antic mode	2	3	4	5	6	7
Basic mode	0	---	---	---	1	2
# color *	1.5	1.5	5	5	5	5
Chars/line narrow wid	32	32	32	32	16	16
Chars/line normal wid	40	40	40	40	20	20
Chars/line wide screen	48	48	48	48	24	24
Scan lines/ pixel	8	10	8	16	8	16
Bits/pixel	1	1	2	2	1	1
Color clocks per pixel	.5	.5	1	1	1	1

Colors

mode 2: Takes the color of PF2 with the lum of PF1
(Artifacting is used very noticeable)
mode 3: Same as mode 2
mode 4: Two bits/pixel in character definitions
00 = BAK 01 = PF0 10 = PF1
11 = PF2 if bit 7 of index = 0, else PF3
mode 5: Same as 4 above
mode 6: Most significant two bits of index
0 = PF0 1 = PF1 etc
mode 7: Same as 6 above

The Graphic Modes

There are 8 graphic modes. Unlike character modes, the values in display memory are not indices into an array of character definitions, but rather are the definitions themselves. Depending on the graphic mode, these values give different results. The following table gives various information about each mode.

Antic mode	8	9	A	B	C	D	E	F*
Basic Mode	3	4	5	6	---	7	---	8
# colors	4	2	4	2	2	4	4	1.5
bytes/line narrow wid	8	8	10	16	16	32	32	32
bytes/line normal wid	10	10	20	20	20	40	40	40
bytes/line wide screen	11	12	24	24	24	48	48	48
Pixels per normal wid	40	80	80	160	160	160	160	320
Scan lines/pixel	8	8	4	4	2	2	1	1
Bits/pixel	2	1	2	1	1	2	2	1
Color clocks per pixel	4	2	2	1	1	1	1	.5

*Mode F values differ when in GTIA modes

Colors

mode 8: Two bits/pixel, 4 pixels/byte
00 = BAK 01 = PF0 10 = PF1 11 = PF2
mode 9: One bit/pixel, 8 pixels/byte
0 = BAK 1 = PF0
mode A: Same as mode 8 above
mode B: Same as mode 9 above
mode C: Same as mode 9 above
mode D: Same as mode A above
mode E: Same as mode A above
mode F: Take the color of PF2 and lum of PF1
(if not in a GTIA mode)

Software and Documentation
©Copyright 1982
Valpar International